DEADLOCK AVOIDANCE AND CONSISTENCY IN DATABASE SYSTEM

AUTHOR(S):

Khanobthamchai, Prasert; Okui, Jun

CITATION:

Khanobthamchai, Prasert ...[et al]. DEADLOCK AVOIDANCE AND CONSISTENCY IN DATABASE SYSTEM. 数理解析研究所講究録 1987, 625: 11-20

ISSUE DATE:

1987-05

URL:

http://hdl.handle.net/2433/99971

数理解析研究所講究録
625巻 1987年 11-20

DEADLOCK AVOIDANCE AND CONSISTENCY IN DATABASE SYSTEM

名古屋工業大学 Prasert Khanobthamchai
名古屋工業大学 奥井 順 (Jun Okui)

1.Introduction.

In database systems, transactions concurrently accessing the database may lead the system to an inconsistent state. The database management system must provide a control mechanism that guarantees completion of transactions and correctness of data used by transactions. Many control mechanisms have been proposed based on the notion of locking,e.g.[4,5]. However, the control mechanism using locking is susceptible to deadlock. Serializability is the criteria of correctness of execution sequences. It is shown that recognizing serializability of an arbitrary execution sequence is NP-complete[3]. However, subset of serializable class of execution sequences can be recognized in polynomial time such as the class DSR[3] (CPSR[2]) and WRW*[7].

Two phase locking policy[4] was shown to guarentee serializability. In the system, serializability requires that each transaction be two phase, i.e. it cannot lock any data after it has unlocked some data. In fact, this control mechanism only checks for availability(unlocked) of data on deciding to grant a lock. Deadlock avoidance in such system can be accomplished by using an algorithm as in the operating system environment. Deadlock avoidance algorithm checks for acyclicity of a directed graph whose edges represent wait-for relations among transactions. However, if an algorithm that checks for acyclicity of graph is used, by changing policy of placing edges in the graph, the two phase restriction can be removed.

In this paper, we propose a scheduler that allows deadlock-free and serializable execution sequences for a locking system. In our model, locking are distinguished into two kinds, shared and exclusive locks. The scheduler determines whether granting a lock will lead the system to an inconsistent state by examining acyclicity of a directed graph. We show that the scheduler allows the largest class of schedules obtainable from locking system. Finally, the class of schedules allowed by the scheduler is shown to be DSR.

2. Model

A formal model of the database and transaction system with ideas on consistency control and deadlock avoidance will be discussed.

2.1 Database.

A database is a set D of entities. A state of the database is an assignment of values to the entities. The consistency constraints of the database define a set CS of consistent states. Generally, the consistency constraints are not explicitly defined and if they are defined, the cost of checking out consistency of the entire database against the constraints would be prohibitly expensive. A transaction is correct if its execution sequence maps any consistent state to a consistent state. For simplicity, we assume that the system starts from a consistent state and each transaction is correct.

Definition. A locked transaction is a finite sequence of steps. Steps of locked transactions are distinguished into three kinds of requests, namely, shared lock (LS.x), exclusive lock (LX.x) and unlock(U.x) request.

A transaction is supposed to issue a shared lock request on an entity before it reads the entity and an exclusive lock before it reads and writes or only writes the entity. Two or more transactions can simultaneously hold shared locks on the same entity and an entity being locked in exclusive mode cannot be locked(in shared or exclusive mode) by any other transaction. Pair of locks on the same entity with one or both locks being exclusive is called incompatible. We assume that a locked transaction neither locks again an entity which is already locked by itself nor unlocks an entity which is not locked and that it unlocks every entity that it locks. Without loss of generality, we assume that each transaction does not request a lock on any particular entity more than once.

Definition. A transaction T has locked an entity x through step i if for some $j<i$ the j^{th} step of T is LS.x (LX.x) and there is no k with $j<k<i$ such that the k^{th} step of T is U.x. A locked transaction is well formed[4] if whenever the i^{th} step of T is an action(read or write) on x, then x is locked through step i.

Provided that T is well formed, the particular time at which T acts (reads or writes) on x is left undefined and can be interpreted as anywhere between the lock-unlock interval. For this reason, the scheduler will be dealing with only sequences of lock and unlock requests. Furthermore, only well formed locked transaction will be considered.

A transaction system τ is a finite set of transactions. A schedule S of τ is an ordering of actions of all transactions in τ which preserves the

ordering of actions of each transaction. A serial schedule is one in which there is no interleaving. From our assumption on transactions, a serial schedule thus preserves consistency of the system. A transaction T_i is said to read an entity x from a transaction T_j(T_j is read on x by T_i) if T_j's write on x in the schedule is followed by read on x by T_i without any write on x by other transaction in between. Two schedules are equivalent if the read-from relations on all entities involved in the two schedules are the same. A schedule that is equivalent to a serial schedule is called a serializable schedule. Apparently, a serializable schedule produces the same effects to the database as a serial schedule.

When a transaction issues a lock request to the system, the system may not grant the lock immediately if the entity is being locked by other transactions with incompatible lock or if granting the lock will lead the system to an inconsistent state or deadlock.

2.2 Serializability and deadlock avoidance in locking systems.

If transactions read an entity without updating value of the entity, the value of the entity left in the database and read by the transactions is not affected by the relative order of reading of transactions. However, if one or more transactions write the entity, the relative order of writing and reading of transactions affects the value of the entity.

Example 1. Let T_1 and T_2 be two transactions having the following steps;
T_1: $\langle LX_1.x, U_1.x, LS_1.y, U_1.y \rangle$, and T_2: $\langle LS_2.x, U_2.x, LX_2.y, U_2.y \rangle$

Sequence of requests granted:

$LX_1.x\ U_1.x\ LS_2.x\ U_2.x$

Possible serializable schedule:

$LX_1.x\ U_1.x\ LS_2.x\ U_2.x\ LS_1.y\ U_1.y\ LX_2.y\ U_2.y$

Example 1 shows that T_1 and T_2 must be scheduled according to the order of their locks on x and their remaining locks on y must be ordered in the same way. This suggests us that the relative orders among all incompatible locks of transactions must be preserved and must be the same. In order to capture this idea, a directed graph can be used. We construct a directed graph $G=\langle V,E \rangle$ whose set of nodes V corresponds to the set of transactions τ and edges represent the orders of locks in schedule S: for any nodes T_i and T_j, include an edge from T_i to T_j if and only if T_i and T_j have incompatible

locks on an entity x and T_i locks x before T_j does in S.

We now show that the graph can be used to obtain an equivalent serial schedule if one exists.

Theorem 1: If graph G is acyclic then schedule S is serializable.

Proof. If the graph is acyclic, by topological sorting we obtain a total order of transactions. Let the sequence of transactions obtained be SR=<T_1 ,...,T_n>, then SR is an equivalent serial schedule for S. We shall prove that if T_i reads some entity, say x, from T_j in S then T_i also reads x from T_j in SR. Since T_j writes x, from our assumption, T_j issues an exclusive lock before writing x and no transaction can lock x while T_j is holding the lock on x. Thus T_i locks x after T_j. There is an edge directed from T_j to T_i in G and thus T_j must be before T_i in SR. Suppose that there is a transaction T_k that writes x. There is an edge between T_k and T_i in G, also an edge between T_k and T_j. There are three possibilities; a) edges directed from T_i to T_k and from T_j to T_k , b) from T_k to T_i and from T_k to T_j , c) from T_j to T_k and from T_k to T_i . The case in which edges are directed from T_k to T_j and from T_i to T_k is impossible since G is acyclic. In a) or b) T_k would have been placed after T_i or before T_j respectively. In c) T_k's lock is between T_i's and T_j's locks thus T_i cannot read x from T_j in S. This is a contradiction. We conclude that T_k cannot be placed between T_i and T_j in SR. Therefore T_i also reads x from T_j in SR. Thus the schedule S and SR are equivalent and S is serializable.

Next, it will be proved that the class of schedule allowed is the largest class obtainable from locking system.

Theorem 2: The class of schedule allowed by graph G is the largest class obtainable from the locking system.

Proof. It suffices to prove that all edges between pairs of nodes in the graph represents the inevitable order between transactions. In order to prove this we use an application of regular expressions. Let x be any entity in D and I={S,X} be the set of symbols where S and X represent shared lock LS.x and exclusive lock LX.x respectively. For any pair of transaction T_i and T_j whose locks on x are incompatible and T_i locks x before T_j, the sequence of locks on x from T_i's to T_j's lock can be described by one of the following regular expressions: a) $X_iI^*S_j$, b) $X_iI^*X_j$, or c) $S_iI^*X_j$. Now let us define a kind of edges as follows: a simple edge on x from T_i to T_j is an edge from T_i to T_j such that T_i and T_j have incompatible locks on x and in the sequence

of locks on x from T_i to T_j there is no exclusive lock on x in between. The situation where there is a simple edge on x from T_i to T_j can be described as a) $X_iS^*S_j$, b) $X_iS^*X_j$, or c) $S_iS^*X_j$. In the first case, T_j reads x from T_i and thus the simple edge specifies inevitable order between T_i and T_j. In the second case, since T_j's lock is exclusive then T_j may have read x from T_i, thus the simple edge between transactions is necessary. In the last case, T_i and T_j may have read x from some transaction, say T_k, thus both transactions must be ordered after T_k in the equivalent serial schedule. Also T_i cannot be ordered after T_j because, otherwise, T_i would not read x from T_k in the equivalent serial schedule. In all cases, we conclude that if there is a simple edge from T_i to T_j then T_i must be scheduled before T_j.

For the general cases above, we use induction on the number m of exclusive locks that appears between locks of T_i and T_j.
Hypothesis: For any $m \geq 0$, T_i must be scheduled before T_j in the equivalent serial schedule. We prove this for case a) $X_iI^*S_j$.
$m = 0$: There is a simple edge from T_i to T_j.
$m > 0$: Let T_l be the transaction whose lock is the $(m-1)^{th}$ X in the sequence, then by induction hypothesis T_i must be scheduled before T_l and there is a simple edge from T_l to T_j, thus T_i must be scheduled before T_j also. Cases b) and c) can be proved by similar arguments. We conclude that edges placed between any pair of incompatible transactions by our policy specifies the inevitable order between transactions and are necessary.

Note that in the case where $m > 0$, the edge from T_i to T_j is not a simple edge and serves for the fact that T_i must be scheduled before T_j. In fact, there is a path of simple edges between every pair of such T_i and T_j and edges that are not simple edges can be omitted.

In operating system, deadlock avoidance is a mothod using predeclared knowledge of requests of processes in examining future resource allocation sequence. In database system, entities can be regarded as system resources on which transactions, upon holding lock, gain right to act. The knowledge of data set to be locked is used to determine future deadlock. Future deadlock can be tested by using directed graph. Nodes of the graph represents the set of transactions and edges represent wait-for relations. A transaction T_i is said to wait for T_j on x if T_j is holding incompatible lock on x when T_i issues lock on x and we include an edge from T_j to T_i in the graph. Apart from the present wait-for relation we also have to consider

future wait-for relations. The necessity of this comes when transactions increasingly lock more entities without unlocking the holding locks and finally end up with some transactions waiting for each other in a cycle. A simple scheme of including future wait-for relation to the graph is to include an edge from transaction that presently holds a lock on an entity to all transactions that have a lock but not yet lock the entity. A more elaborate scheme where edges are placed only between transactions that is probable to cause deadlock can be exercised, see [1]. However, the latter one may not help in our model where even the edges placed by the simple scheme must be present in serializing the schedule.

Lemma 1. Let P be the sequence of lock granted so far. Let $G=\langle V,E\rangle$ be directed graph whose nodes V represent the set of transaction τ and edges E_1 and E_2 constructed as follows:

For any pair of T_i and T_j and any entity $x \in D$ such that T_i and T_j have incompatible locks on x.

E_1: a) If T_i's lock on x is before T_j's lock in P, an edge is included from T_i to T_j.

b) If T_i is holding a lock on x or have already locked x(T_i's lock is in P) and T_j has not locked x, an edge is included from T_i to T_j.

E_2: If T_i is holding a lock on x and T_j has not locked x, an edge is included from T_i to T_j.

then if there is an edge E_2 constructed from T_i to T_j then there is an edge E_1 from T_i to T_j.

The argument is apparent from construction of edges.

3. Algorithm

Our algorithm examines the state when a lock is supposed to be granted and repeatedly finds transactions that can be advanced forward until finish until no more transactions. The lock can be granted if all transactions can be advanced to finish.

Since we have to construct directed graph from the partial schedule P repeatedly when examining whether to grant a lock, the algorithm keeps the graph instead of the list P. We assume that on arriving in the system, each transaction T_i submits a set L(i) of locks to be issued. Each element in L(i) has the form $LX_i.x$ or $LS_i.x$ where x is the entity to be locked. Subscripts of lock requests are omitted when the transaction concerned is

obvious. For each entity $x \in D$, we associate two variables count(x) and mode(x). The variable count(x) is used for counting the number of transactions presently holding lock on x and mode(x) signifies the mode in which x is locked which is exclusive(mode(x)=X) or shared(mode(x)=S). Also with each entity, we keep a set LL(x) of last locks which is the set of lock requests that has locked x counting from the last exclusive lock(the last exclusive lock on x is also included).

Algorithm (SCHEDULER)

0: [Initialization] FAILS := true. NEW := false.
For all x in D, count(x) := 0, LL(x) := ϕ.
Let G=<V,E> be a directed graph and G := ϕ, V := ϕ.

1: [Wait for next request]
Let q be the request arrived and x be the entity associated with q and T_i be the transaction issuing q. NEW:=true.

2: If $T_i \cap V = \phi$
then V := V $\cup$ T_i,
for all elements p in L(i) and element $x \in D$ and T_j in V,
do one of the followings.
a) p = $LX_i.x$:
for all elements r in LL(x),
if r = $LS_j.x$ or r = $LX_j.x$ then include an edge from T_j to T_i.
b) p = $LS_i.x$:
for all elements r in LL(x),
if r = $LX_j.x$ then include edge from T_j to T_i .

3: Do one of the following.
a) q = $LS_i.x$ or q = $LX_i.x$:
If NEW then delete q from L(i) else delete q from DEL.
Test q by the algorithm CHECK.
If FAILS then mark q, augment DEL with q
else change G to G', discard old version of G,
do one of the followings,
a) q = LX.x: mode(x) := X, count(x) := 1, LL(x) := {q}.
b) q = LS.x: mode(x) := S, count(x) := count(x) + 1,
LL(x) := LL(x) $\cup$ {q}.
If NEW then goto 1
else pick up new q from unmarked elements of DEL,

If no such elements then goto 1 else goto 3.

b) q = U_i.x:

count(x) := count(x)-1.

repeat finding source node T_j in V such that L(j) = ϕ and there is no element of the form LX_j.y or LS_j.y, where y $\in$ D, in DEL, delete T_j from V all edges directed from T_j until no such source node.

If count(x) $\neq$ 0 then goto 1

else unmark all element in DEL, NEW:=false,

pick up new q from unmarked elements in DEL

if no such elements then goto 1 else goto 3.

4. End.

CHECK

1: If (q=LX.x and count(x)=0) or (q=LS.x and (mode(s)=S or count(x)=0)
then FAILS:=true, return.

2: Construct G' from G by copy G to G',

for all $T_j \in V - T_i$,

if ((T_j has LS.x or LX.x in L(j) or DEL) and q = LX.x) or
((T_j has LX.x in L(j) or DEL) and q = LS.x)
then include an edge from T_i to T_j.

3: If G' is acyclic then FAILS := false else FAILS := true, discard G'.

4: Return.

Informally, the algorithm works like this: After a request arrived in the system, if the request is from a new transaction, the algorithm updates the graph to include the transaction and necessary edges to it(step 2). Then, the algorithm checks if the request can be granted immediately, if this fails it puts the request into the delay list, otherwise, it grants the locks and changes the graph to a new version. Upon receiving an unlock request the transaction checks to see if any delayed requests can be granted and returns to wait for a new request. The algorithm 'CHECK' creates a new version G' of G which represents the situation when the request is supposed to be granted. On doing this it first includes all new necessary edges which should be directed from the transaction having the lock which is now a new holding lock to all transactions that have not locked the entity involved.

It should be noted that edges from transactions holding locks to transactions having unexecuted locks are placed in the graph when G' is

constructed and when a new transaction arrived in the system. However, all edges from E_1 defined in lemma 1 are not completely placed in the graph but all simple edges, which we have seen to be necessary, are placed in the graph. Paths of simple edges function like those missing edges.

Theorem 3: Graph G is acyclic if and only if partial schedule P can be extended to a complete serializable schedule.

Proof. 'If' Since the graph is acyclic then it can be topological sorted and thus we obtain a sequence of transactions. By deleting source node and edges directed from the source node repeatedly until no node is left is equivalent to have transactions execute their remaining locks one by one until all transactions finish. Since the source node is not directed by any edges, all of its remaining locks can be executed without violating consistency and all entities can be locked since the transaction that previously holding lock on the entities has completed and thus unlocked the entities.

'only if' We prove that if the graph is cyclic then either there will be deadlock or inconsistency. If the graph is cyclic then each transaction participating in the cycle is directed by edges from some transactions and therefore its remaining locks must be either executed after some transaction's lock or some entity is being locked by other transactions. Thus, transactions participating in cycle can not be extended to completion.

Theorem 4: The class of schedule allowed by the algorithm is DSR.

Definition. For any transaction T_i and T_j, and any entity $x \in D$

1. RW-constraint: If T_i reads x before T_j writes x then T_i must be serialized before T_j.
2. WR-constraint: If T_i writes x before T_j reads x then T_i must be serialized before T_j.
3. WW-constraint: If T_i writes x before T_j writes x then T_i must be serialized before T_j.

The class of schedule conforms to these constraints is DSR.

Proof. Each edge in the graph is placed between pair of nodes representing transactions having incompatible lock. There are 3 situations in which T_i has an outgoing edge directed to T_j because they have incompatible lock on an entity x; T_i has a shared lock and T_j has an exclusive lock, T_i has an exclusive lock and T_j has a shared lock or both T_i and T_j have exclusive locks. These cases correspond to T_i reads x and T_j writes x, T_i writes x and T_j reads x, and both transactions write x, respectively. Since our

locking policy does not allow T_i and T_j to lock x simultaneously, the order of T_i and T_j's actions(read or write) on x depends on the order of their locks on x. Since an edge is directed from T_i to T_j if T_i locked x before T_j in P or T_i has already locked x and T_j hasn't locked x, in either case, T_i's lock on x is before T_j's lock. Thus T_j's action on x is before T_j's action. Edges in the graph imposes constraints to schedule as same as constraints in class DSR.

4. Conclusion.

In this paper, we propose a scheduler for a locking system that checks for serializability by examining acyclicity of a directed graph. The scheduler allows a class of schedules that is shown to be DSR. Deadlock avoidance algorithm was adapted for the purpose. However, in the deadlock problem, wait-for relations goes away after transactions release entities locked but in serialization of schedule constraints of serialization does not. We have seen that when a more complex(and time-consuming) algorithm is used concurrency can be improved. However, this is the matter of tradeoff on issue of concurrency and efficiency.

REFERENCES:

1. SUGIYAMA,Y.,ARAKI,T.,OKUI,J.,AND KASAMI,T. Complexity of the deadlock avoidance problem. Trans. Inst. of Electrom. Comm. Eng. Japan J60-D,4 (Apr. 1977),251-258.
2. BERSTEIN,P.A.,SHIPMAN,D.W.,AND WONG,S.W. Formal aspects of serializability in database concurrency control. IEEE Trans. Softw. Eng. SE-5,3, (1979),203-216.
3. PAPADIMITRIOU,C.H. The serializability of concurrent database updates. J.ACM 26,4(Oct. 1979),631-653.
4. ESWARAN,K.P.,GRAY,J.N.,LORIE,R.A.,AND TRAIGER,I.L. The notions of consistency and predicate locks in database system. Commun. ACM. 19,11 (Nov. 1976),624-633.
5. YANNAKAKIS,M. A theory of safe locking policies in database system.J.ACM. 29,3(Jul. 1982),718-740.
6. AHO,A.V.,HOPCROFT,J.,AND ULLMAN,J. The design and analysis of computer algorithm. Addison-Wesley,Reading,Mass.,1974.
7. KATOH,N.,IBARAKI,T., AND KAMEDA,T. A cautious transaction scheduler with admission control. ACM trans. database syst. 10,2(Jun 1985),205-229.